UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2019

Test #1 (revised)

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 5 problems on 12 pages. Officially, it is worth 17 points (out of a total of 200).

This is an open-book test. You have 110 minutes to complete it. You may consult any books, notes, or other non-responsive objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and TA in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the outside of the building once or twice.

Your name: _____ Login: _____

Login of person to your Left: _____ Right: _____

Discussion TA: _____

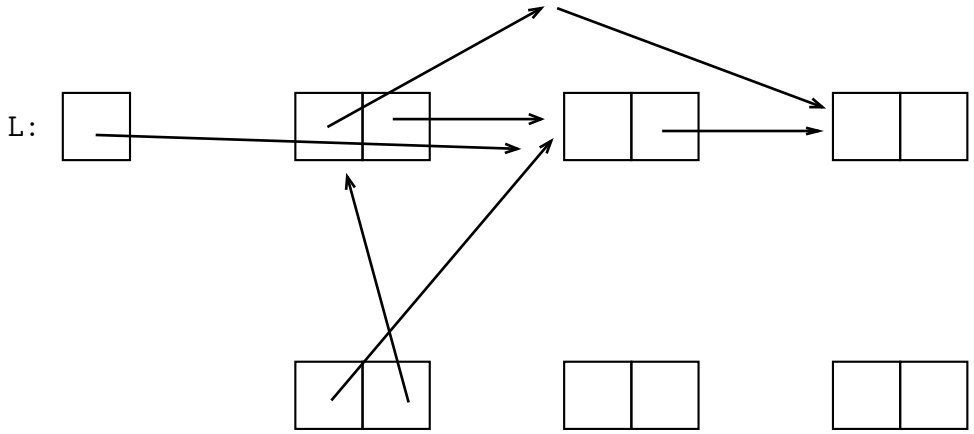
Reference Material.

```
/* arraycopy(FROM_ARR, FROM_INDEX, TO_ARR, TO_INDEX, LENGTH) */
import static java.lang.System.arraycopy;

public class IntList {
    /** First element of list. */
    public int head;
    /** Remaining elements of list. */
    public IntList tail;

    /** A List with head HEAD0 and tail TAIL0. */
    public IntList(int head0, IntList tail0)
    { head = head0; tail = tail0; }

    /** A List with null tail, and head = HEAD0. */
    public IntList(int head0) { this(head0, null); }

    /** Returns a new IntList containing the ints in ARGS. */
    public static IntList list(Integer ... args) {
        // Implementation not shown
    }

    /** Returns true iff L (together with the items reachable from it) is an
     *  IntList with the same items as this list in the same order. */
    @Override
    public boolean equals(Object L) {
        // Implementation not shown
    }

    /** Return the length of the non-circular list headed by L. */
    public int size() {
        // Implementation not shown
    }

}
```

1. [4 points]

(a) [2 points] Fill in the box and pointer diagram to show the variable `L` and the objects and their contents resulting from a call to `create()` when program execution reaches the point labeled `HERE`, using the empty boxes provided. Some objects may be unreachable from any named pointer variable (may be "garbage" to use the technical term); ***show them anyway.*** The double boxes represent array objects. You need not use all the boxes.

```
static Object[] M(Object left, Object right) {
    return new Object[] { left, right };
}

static Object[] A(Object obj) {
    return (Object[]) obj;
}

static void create() {
    Object[] L;

    L = M(null, M(null, M(null, null)));
    L = M(A(L)[1], L);
    A(A(L)[1])[0] = A(A(L)[0])[1];
    L = A(A(L)[0]);
    // HERE
}
```

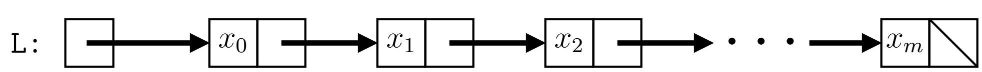


(b) [2 points] For this one, fill in the program so that it converts the *Before* diagram to the *After* diagram, given the variable `L`. Here, `L` is a linked list of some unspecified number of integers (type `IntList`) containing the (unspecified) values x_0, $x_1, \ldots$. The idea is to (destructively) insert new `IntList` objects between the original existing `IntList` items such that each new item contains the average of the two surrounding values. The resulting list must continue to contain all the original objects plus the newly inserted ones.

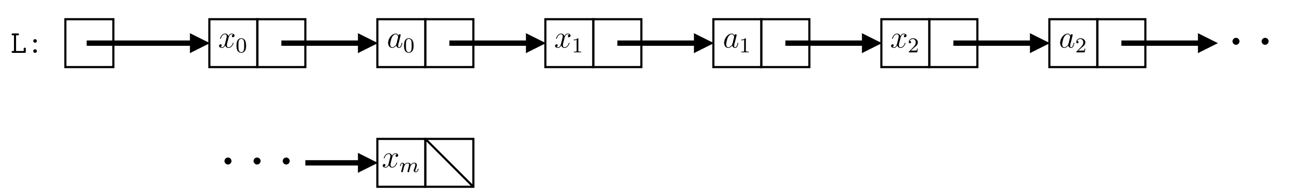
For example, if `L` initially points to a list containing `[0, 2, 10, 8, 15]`, it would end up pointing to a list containing `[0, 1, 2, 6, 10, 9, 8, 11, 15]` and the `IntList` objects containing 0, 2, 10, 8, and 15 would be reused in this result.

```
assert (L != null);  /* You may assume this. */

for (__IntList l = L__________; l != null__________; l= l.next.next_______) {

    L.next =__________ = new LinkedList(l.head+l.next.head/2, l.next.next)______

                         _______________________________________________;
                         // Use the cotinuation line above if needed.
}
```

Before:



After:



where $a_i = \lfloor (x_i + x_{i+1})/2 \rfloor$

2. [6 points] Assume that there is an interface named `Pred` that is intended to represent true/false functions (predicates) on integers.

```
public interface Pred {
    /** Return true iff X satisfies this predicate. */
    boolean test(int x);
}
```

(a) Fill in the implementation of `extract` on the next page to fulfill its comment. The function is intended to remove elements from one array that satisfy a given predicate and move them to another, returning the number of elements moved.

Thus, if initially an array `Q` contains

```
[ 5, 2, 0, 6, 9, 10, 12, 13 ],
```

the array `R` contains arbitrary values:

```
[ ?, ?, ?, ?, ? ],
```

and the `Pred` object `isOdd` is satisfied by exactly the odd numbers, then the call `extract(Q, R, isOdd)`, will return 3, and the contents of the arrays will be

```
Q = [ 2, 0, 6, 10, 12, ?, ?, ?]
R = [ 5, 9, 13, ?, ? ]
```

(Here, '?' denotes a value that is unspecified.)

```
// (Repeated from previous page.)
public interface Pred {
    /** Return true iff X satisfies this predicate. */
    boolean test(int x);
}


/** Move all values from FROM that satisfy the predicate P into
 *  TO in their original order.  Move the remaining items in FROM
 *  to the left so as to fill in the places whose values were moved.
 *  Return the number of values that were removed from FROM.
 *  Assumes that the length of TO is sufficient to hold all the
 *  values to be moved. */
public static int extract(int[] from, int[] to, Pred p) {

    int toInd, fromInd, nextEmptyFrom;

    toInd = fromInd = nextEmptyFrom = 0__________________;

    while (_fromInd < from.length_____________________________) {

        if (_p.test(from[fromInd])_____________________) {

            __to[toInd], from[fromInd]__ = _to[toInd], from[fromInd]_____;

            _____________ += __________;
        } else {

            ____________________ = ________________________;

            _____________ += __________;
        }

        ______________________________________;
    }

    return _________________________________;
}
```

(b) Using `extract`, fill in the definition of `arrPartition` so that when `arrPartition(A, B)` returns, all values that are $\leq$ `A[0]` are in `A` and all those $>$ `A[0]` are in `B`.

```
// (Repeated from previous page.)
public interface Pred {
    /** Return true iff X satisfies this predicate. */
    boolean test(int x);
}


/** Move all values in LEFT that are > LEFT[0] into RIGHT, and
 *  fill in the resulting gaps in with the remaining elements,
 *  returning the number that were moved. */
public static int arrPartition(int[] left, int[] right) {

    return extract(right , left, new Pred(right[0])                    );
}

// In a separate file:

class Pred                                  {
        public int reference;
    ------------------------------------------------------------------
      public pred(int x){
    ------------------------------------------------------------------
        reference = x;
    ------------------------------------------------------------------
      }
    ------------------------------------------------------------------
      @Override
    ------------------------------------------------------------------
      boolean test(int x){
    ------------------------------------------------------------------
        return x>reference
    ------------------------------------------------------------------
      }
    ------------------------------------------------------------------

    ------------------------------------------------------------------

    ------------------------------------------------------------------

}
```

3. [1 point] What do Mt. Blanca, Mt. Taylor, the San Francisco Peaks, and Mt. Hesperus have in common?

4. [4 points] The function `partition` performs a step used in a sort procedure. Given an `IntList` `L` such as

```
[10, 5, 12, 1, 3, 20, 0]
```

the call `partition(L)` will rearrange the values in `L` so that all values less than or equal to the first (10 in this case) will come first in the list, before all those greater than the first. This first value is called the *pivot*. For example, for `L` above, the result could be

```
[0, 3, 1, 5, 10, 20, 12 ].
```

For the particular way we partition in this problem, the portions of the result that are $\leq$ the pivot or $>$ the pivot are in the reverse order from how they appeared in the original list `L`.

We use a helper function called `part`. This takes a pivot, a list `L` of remaining items to be partitioned, two `IntList` objects—`leftFirst` and `rightLast`—that are the first and last objects in a list of values $\leq$ the pivot, and an `IntList` object `right` that is the first object in a list of values $>$ the pivot (or null if there are none). It then adds each element of `L` in front of `leftFirst` or `right`, and finally returns the result of concatenating the two lists. Hence, if `L` contains `[1, 3, 20, 0]`, `leftFirst` contains `[5, 10,...]`, `leftLast` points at the second item (10) in the list `leftFirst`, and `right` contains `[12]`, then `part(10, L, leftFirst, leftLast, right)` will return

```
[0, 3, 1, 5, 10, 20, 12 ].
```

(The list starting at `leftFirst` is allowed to contain additional stuff after the item pointed to by `leftLast`, but it gets thrown away in the result.)

Fill in the definition of `partition` to meet its specification.

```
/** Given that L consists of the values x1, ..., xn, where n >= 1,
 *  return a list containing the same values, but with all values that
 *  are <= x1 coming before all values that are > x1.  The operation
 *  is destructive; no new IntList objects are created. */
public static IntList partition(IntList L) {
    return part(L.head, L.tail, L, L, null);
}

/** Assuming that LEFTFIRST and LEFTLAST are the (non-null) first
 *  and last IntList objects in a list, (destructively) return the
 *  result of adding all elements of L that are <= PIVOT to the
 *  front of LEFTFIRST, all elements > PIVOT to the front of RIGHT,
 *  and then concatenating the elements in RIGHT after those from
 *  LEFTFIRST to LEFTLAST. The initial .tail of  LEFTLAST
 *  is overwritten. */
private static IntList part(int pivot, IntList L,
                            IntList leftFirst, IntList leftLast,
                            IntList right) {
    if (_____) {
        _____ = _____;
        return leftFirst;
    }
    IntList next = L.tail;

    if (_____) {
        L.tail = _____;
        return part(pivot, _____);
    } else {
        L.tail = _____;
        return part(pivot, _____);
    }
}
```

L == null

LeftLast.tail

right

L.item >pivot

L.tail.tail

l.tail, leftfirst, leftlast,

5. [3 points] What is printed by running `java C`? Assume that the function `test1.utils.PRT(...)` acts like Python's `print` function and prints its arguments on a line separated by blanks. There are no compilation or runtime errors in this code.

```
import static test1.utils.PRT;

class A {
    int x = 1;
    static int y = 2;
    void f() { PRT("A:f()", x, y); }
    static void g() { PRT("A:g()", y); }
    void h(A x) { PRT("A:h(A)", x.x, x.y); }
}

class B extends A {
    int x = 3;
    static int y = 4;
    void f() { PRT("B:f()", x, y); }
    static void g() { PRT("B:g()", y); }
    void h(A x) { PRT("B:h(A)", x.x, x.y); }
}

public class C extends B {
    int x = 5;
    static int y = 6;
    void f() { PRT("C:f()", x, y); }
    static void g() { PRT("C:g()", y); }
    void k(A x) { PRT("C:k(A)"); x.f(); ((B) x).g(); }

    A anA = new A();
    B aB = new B();

    public static void main(String[] unused) { new C().doStuff(); }
    void doStuff() {
        PRT("Line 1"); anA.f(); f(); ((B) this).f();
        PRT("Line 2"); this.g(); C.g(); aB.g(); ((A) this).g();
        PRT("Line 3"); aB.h(anA); h(anA);
        anA = (A) aB;
        PRT("Line 4"); anA.f(); anA.h((A) aB); anA.h(this);
        PRT("Line 5"); anA.g(); aB.h(anA); aB.h((B) anA);
        PRT("Line 6"); this.k(anA);
    }
}
```

```
Line 1

----------------------------

----------------------------

----------------------------

Line 2

----------------------------

----------------------------

----------------------------

----------------------------

Line 3

----------------------------

----------------------------

----------------------------

Line 4

----------------------------

----------------------------

----------------------------

Line 5

----------------------------

----------------------------

----------------------------

Line 6

----------------------------

----------------------------

----------------------------
```